Cloud Computing Aplications and Services - Pratical Assignment

Automatic Installation and Configuration of Laravel Application

29 of December, 2023

Group 20

Bernard Georges
University of Minho
pg53698

José Pedro Fonte
University of Minho
a91775

Abhimanyu Aryan
University of Minho
pg51632

Miguel Silva
University of Minho
pg54097

José Carvalho
University of Minho
pg53075

Abstract

This project aims to deploy the Laravel.io application. A online platform serving as the central hub for the PHP framework laravel. This document will elucidate the intricacies of the deployment methodology, shedding light on the tools utilized to ensure a seamless and efficient launch of this dynamic web application.

1 | Introduction

The deployment of the Laravel.io application, a central hub for the Laravel PHP framework, represents a significant endeavor in cloud computing and application scalability. This report details our journey in deploying Laravel.io on Google Kubernetes Engine (GKE), a cloud-based environment that offers advanced scalability and management features for containerized applications.

Our primary objective was to evaluate the application's performance under stress, specifically within the GKE infrastructure. By leveraging JMeter, we were able to simulate various load conditions and monitor how the application behaves under stress. This allowed us to identify potential performance bottlenecks and scalability issues.

A key aspect of our project involved analysing the scaling the application in response to varying loads. Using the capabilities of GKE, we implemented a strategy to increase the number of replicas of the application's pods when under heavy load, as detected by JMeter tests. This approach ensured that Laravel.io could maintain optimal performance and availability, even during peak usage.

This report chronicles our experiences, methodologies, and the tools we used, providing a comprehensive overview of the deployment process and the challenges encountered in optimizing Laravel.io for a cloud-based infrastructure like GKE.

2 | Architecture and main components

We have identified two main components of the Laravel application.

The first one is related to the front-end of the application and is essentially the entry point of the application, since it receives and responds to clients.

The other service we have recognized is the *MySQL* service. As the name suggests, this component manages all the data for the application using *MySQL* as the database engine. This service only answers requests from the front-end service.

3 | Tools

The deployment and configuration of the Laravel.io application are executed using a suite of sophisticated tools, each chosen for its specific advantages in handling cloud-based applications.

1. Docker: Docker plays a crucial role in our deployment strategy. It provides a containerization platform that enables us to package the Laravel.io application along with its dependencies into a container. This containerization ensures consistency across multiple development and release cycles, supporting our application's portability and scalability.
2. Ansible: Ansible is utilized for automating the deployment and configuration process. It offers a simple yet powerful automation engine that helps in managing complex deployments with ease. In our project, Ansible is used to orchestrate the deployment and undeployment of the Laravel.io application and its environment on the cloud infrastructure.
3. Kubernetes and Google Kubernetes Engine (GKE): Kubernetes is integral to our approach, providing a platform for automating the deployment, scaling, and operation of application containers. We use GKE to leverage Google's infrastructure for managing Kubernetes clusters. Kubernetes also facilitates load balancing and service discovery, essential for handling dynamic user demands.
4. Persistent Volume Claims (PVCs): PVCs in Kubernetes are used for managing storage in our application. They allow us to abstract the details of how the storage is provided and how it's consumed. This is critical for Laravel.io, particularly for database storage and persistent data management.
5. JMeter: JMeter is a tool used for performance testing. It allows us to conduct extensive stress tests on the application, enabling us to evaluate its performance under various loads. This ensures that Laravel.io can handle high traffic and usage, maintaining stability and responsiveness.

4 | Installation and configuration of the application

This chapter delves into the installation and configuration process of the Laravel.io application on Google Kubernetes Engine (GKE). Leveraging Docker for containerization and Ansible for automation, we go through the key steps in deploying Laravel.io with a focus on maintaining consistency and scalability.

Figure 1: Repository Structure regarding Docker and Ansible

4.1 | Docker

The repository's Docker section follows the structure outlined in Figure 1. Within this structure, the group is comprised of five distinct versions: 0.1 (first checkpoint version), 0.2, 0.3 (intermediate versions), and 0.4, 0.5 (regarded as the current versions). Notably, the differences between these versions are as follows: Versions 0.2 and 0.4 perform database seeding, while versions 0.3 and 0.5 do not. Additionally, versions 0.2 and 0.3 make changes to the file config/session, whereas versions 0.4 and 0.5 do not exhibit such modifications.

All mentioned Docker image versions are stored on our Docker Hub repository.

All different Dockerfiles created are using Ubuntu 22.04 OS as a base. Then, we install vital dependencies, configure PHP 8.2, and add Composer, Git, Node.js, and npm. Laravel.io's repository is then cloned into /app, script.sh is copied(in v0.2/0.4), and user permissions adjusted. Composer dependencies are installed, script.sh handles tasks like database seeding (v0.4) and Laravel application is launched.

4.2 | Ansible

Ansible automates the deployment of the Laravel.io application through the use of roles and playbooks. By organizing tasks into roles we streamline the installation and configuration process, ensuring consistency and reproducibility across different environments.

4.2.1 | Default Playbook

- gke-cluster-create.yml: Ansible file for creating GKE clusters.
- gke-cluster-destroy.yml: Ansible file for destroying GKE clusters.
- laravelio-deploy.yml: Ansible file for deploying Laravel.io.
- laravelio-undeploy.yml: Ansible file for undeploying Laravel.io.
- test-all.yml: Ansible file for running all tests.
- gcp.yml : Ansible file that includes every variable used for the creation and destruction of the clusters and deployment, undeployment and testing of the Laravel.io application.

4.2.2 | Roles

The `Roles/` directory contains the following roles: `deploy_files`, `gke_cluster_create`, `gke_cluster_destroy` and `test_laravelio`. The `deploy_files` role contains every yml file made by ourselves. This directory plays a central role in managing deployment files for Laravel.io, orchestrating key configurations such as deployment, database seeding, service setup and undeployment. This role ensures the smooth execution of deployment tasks, facilitating a well-organized deployment lifecycle.

The `gke_cluster_create` role focuses on automating the creation of a Google Kubernetes Engine (GKE) cluster, streamlining the setup of essential infrastructure for Kubernetes clusters on Google Cloud. On the contrary, the `gke_cluster_destroy` role is tailored for the efficient teardown of GKE clusters, simplifying the process of deprovisioning resources and cleaning up infrastructure.

Finally, the `test_laravelio` role oversees testing scenarios for the Laravel.io application, encompassing access testing, login functionality testing, and a comprehensive testing suite. Together, these roles contribute to a structured, automated approach for deploying, managing, testing, and undeploying the Laravel.io application.

5 | Questions

1.a For a growing number of clients, which application components could be a performance bottleneck?

Answer: Front-end service is the most critical one, because it services every request, while *MySQL* service is not needed for some types of requests, for example, getting the login page. Even though this does not mean that *MySQL* service is not a bottleneck, because usually database operations takes some time, so we should pay attention to it as well. But, as mentioned, when we have thousands of users using *laravelio* website, most of them certainly will not be executing operation with the database, but all of them will be interating with the web interface.

To sum up, both components represent a bottleneck, but front-end is likely to be the most critical because the ratio of request per second is way higher when compared with database component.

1.b How does the application perform against different numbers of clients and workloads?

Answer: Overall our application is really slow, since we only have 1 pod for each component. We made the same automated tests used for checkpoint #2, and they are:

1. *GET* request to the main page
2. *GET* request to the login page
3. *POST* request to login
4. *GET* request to user page

We tested the application for the following amounts of clients:

1. 1000
2. 100
3. 10

All of the clients execute the listed tests. We noticed that for 10 clients, the website was able to responde but it took some seconds, but for one hundred and one thousand clients, the website was taking too much time to respond. We further detail these tests and their results on section 7 and 8.

1.c Which application components could constitute a single point of failure?

Answer: Actually both of them can represent a point of failure.

MySQL component is a single points of failure for operations. For example, getting a user page and authenticating into the website requires to acess the database, and these tasks will not work if the *MySQL* service is down.

The front-end component is also a single point of failure, since if it is down the rest of the application will not work.

2.a What load distribution/replication optimizations can be applied to the base installation?

Answer: To optimize the load distribution we opted to create a few replicas of the application. This in threory would help reduce the overload as the workload would be distributed among the replicas.

2.b What is the impact of the proposed optimizations on the performance and/or resilience of the application?

Answer: To check the impact of the optimizations mentioned we ended up using JMeter's benchmarking tools. We made multiple HTTP requests and using the data that JMeter captured, we created graphs. The first ones were made using only one replica, whilst the latter two were made for four replicas.

As noted in the graphs from section 7, both the latency, and elapsed time per request decreases significantly when we increase the number of replicas. This is interesting because even the requests that take the longest time still take less than the a normal request with one replica, taking in some cases a third of the time.

Another conclusion we can take from the analysis of these graphs is that the latency and elapsed times increase with the increase with the amount of bytes sent and recieved, this makes total sense as physical resources may not be able to handle information pass a certain threshold.

So, with this optimization, we managed to improve the scalability of the app, which is one of the pillars of distributed systems.

We could have also taken a look at more metrics, but we believe these are enough to prove the efficiency gain acquired through this technique.

To better undestand other means through which we performed benchmarking, we have the next section in of the document which is entirely dedicated to this topic.

6 | Monitoring

In order to enhance the website's overall performance, the application underwent rigorous stress tests. Throughout these assessments, diligent monitoring was conducted leveraging Google Cloud monitoring tools to assess and optimize key performance metrics.

The Google Cloud platform provides a plethora of metrics for monitoring various aspects of deployed applications. In the context of deploying the Laravel.io application, our focus primarily rested on key performance indicators such as CPU utilization, memory usage, and disk usage. These metrics served as pivotal indicators of the application's health and performance.

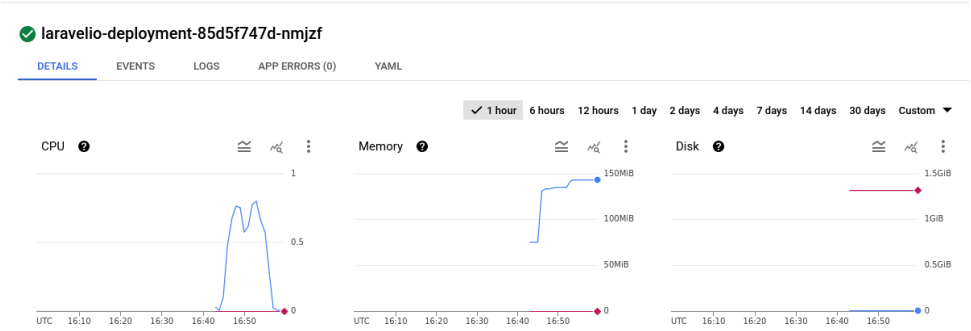

Figure 2: Google Cloud view of the metrics.

As depicted in the above metrics, spikes indicate a notable surge in performance across the aforementioned elements. While these spikes signify an improvement, it's crucial to note that reaching critical usage levels can potentially lead to performance drops for users. This emphasizes the need for a careful balance in optimizing performance without compromising the user experience during peak usage scenarios. This paired with the evaluation metrics helped the group to improve the load balancing and performance of the distribution.

7 | Evaluation

7.1 | Tests

As previously stated, we chose to run the tests provided by the professors for the previous stage. These tests in our opinion are sufficiently good as they end up giving us multiple interesting results. This is due to the tests being related to specific parts of the program, we test the frontend, we test operations that utilize the database and we test both HTTP *GET* and *POST* requests that represent two different kinds of actions handled by the application.

We tested our cloud solution for both one and multiple (4) replicas, and when it came to the number of threads specified on the JMeter's interface, for one replica we only tested ten clients, mostly because

it is ideal to have more replicas and the response time when compared to more replicas would always take longer, for four we ended up testing more clients as a way to study scalability.

7.2 | Evaluation Graphics

Based on the data that JMeter captured, we were able to build some graphics. Now we are going to mention them, explaining the information that they have and explaining why we have used them.

1. Average Response Time per Page: This graphic compares the average response time per page. The main point of this metric is to check if there is any page that responds faster or slower than others to understand possible bottlenecks and/or find out the more complex parts of the system.
2. Latency Histogram: As histogram that shows the absolute frequencies of each time interval. The reason we decided to look at this was to check how much time our service is taking to respond.
3. Response Time : This graphic has the same information as the last one, but it represents the data differently. Instead of the bars, we can compare what are the common response time values for different ammounts of clients / threads. The intention is to catch outliers and, as already said, analyse what are the most common values for response time.

7.3 | Results

7.3.1 | 1 replica - 10 Clients

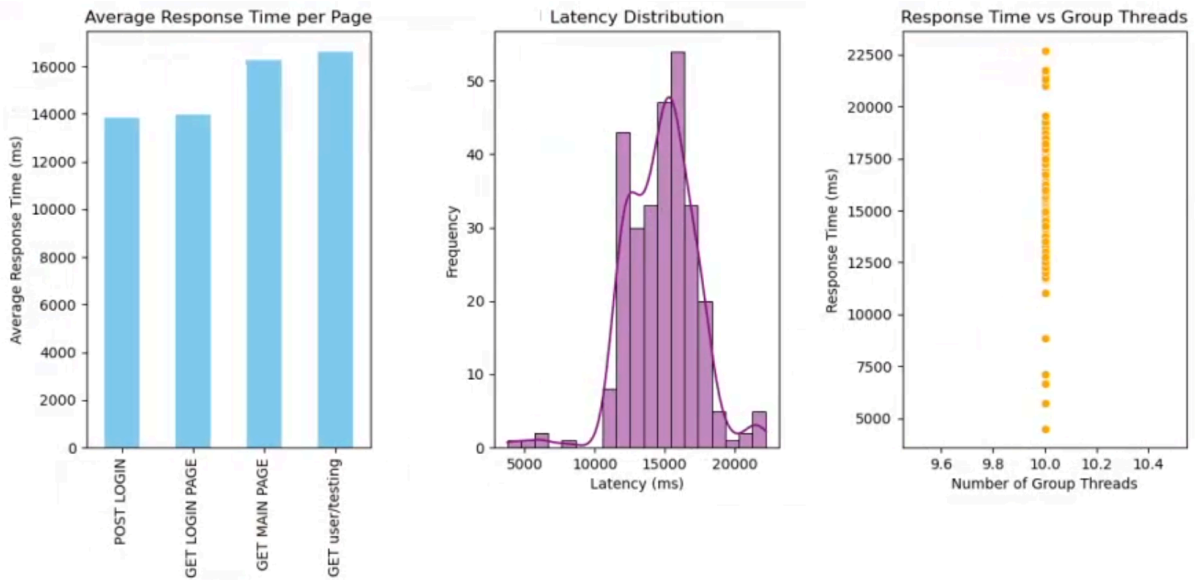

Figure 3: Values regitered with 10 threads and 1 replica

In relation to the scenenario where we only have 1 replica, we can conclude that:

- There is not much difference between the average time per request.
- Login request takes less time compared with other requests.
- The most common response time is between 12.5 and 20 seconds.

As we can see, this version is not really great, because if there is an application that takes more than 10 seconds to respond most users would drop it.

7.3.2 | 4 replicas - 100 Clients

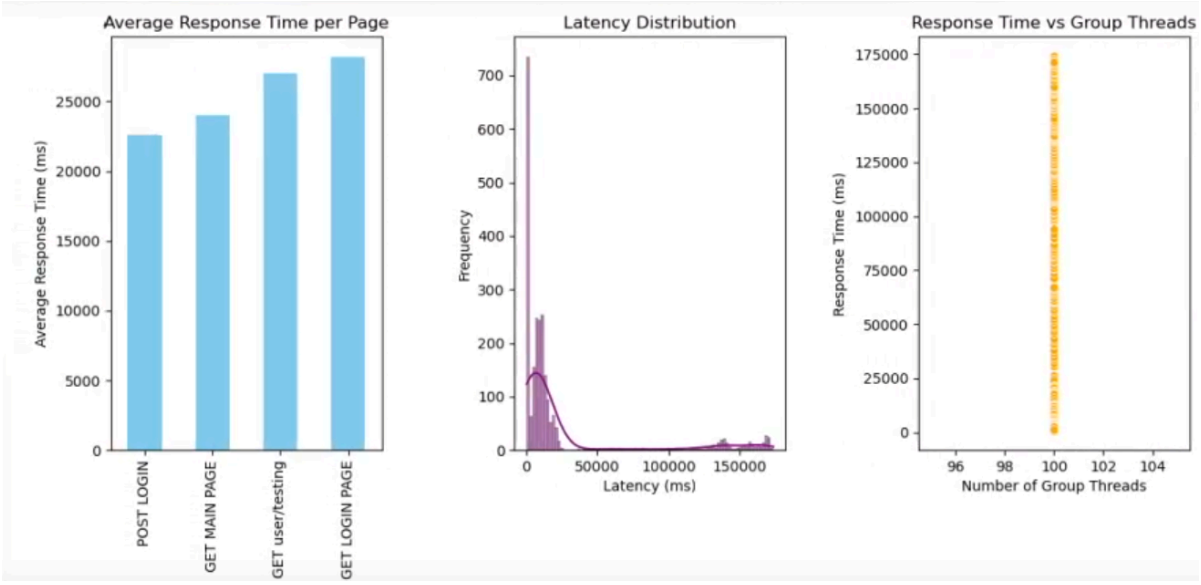

Figure 4: Values regitered with 100 threads and 4 replicas

Here we have 4 replica and we have 100 clients making requests, we can conclude that:

- There is not much difference between the average time per request.
- The most common response time is between 20 and 30 seconds, but we can see from the second graph, that most of our request have a really low latency.

As we can see, this version is not perfect, but it is still better than the previous one, since we are attending 100 clients instead of 10 clients. It is important to note, that the latency was below 1 second when we were working with 10 clients, so we didn't measure performance values for 10 clients.

7.3.3 | 4 replicas - 1000 Clients

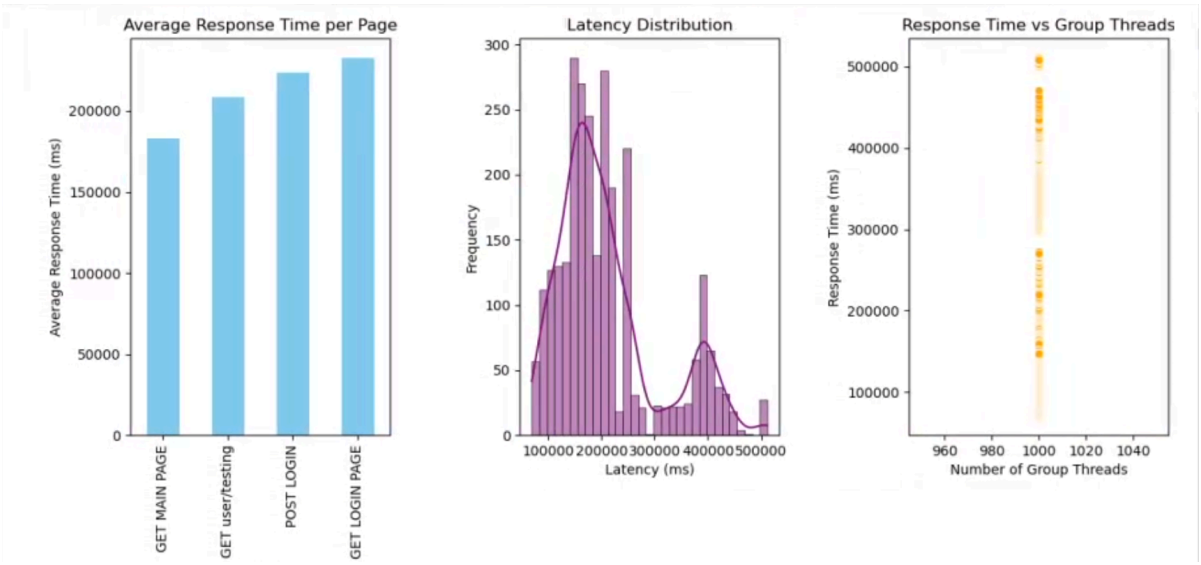

Figure 5: Values regitered with 1000 threads and 4 replicas

For 4 replicas and 1000 clients making requests, we can conclude that:

- There is not much difference between the average time per request.
- The most common response time is between 15 and 25 seconds.

With these results we can conclude that our solution does not support 1000 clients, since the application gets way too slow to be used in a real context.

8 | Conclusion

After concluding this project, we can clearly state that our understanding of the subject's topics grew exponentially. Mostly because we had a way to practise what we learned in the classes on our own in a new application.

Another positive of this project is that we worked with cloud directly in ways we had never done before, we consider that to be a realy good thing because we never know when we will need to apply what we learned in the industry. The cloud is the future and we have to ready to embrace it.

Still, during this project we had our fair share of issues which we explained in this document, even so we can guarantee that if we did not have the GitHub FaQs with the professors constantly responding to our posts we would have had many more. This was a complex project plagued with issues, which is normal, these are sophisticated technologies that require a certain amount of expertise. Therefore we have nothing but gratitude for the support provided even if we believe we could have had more clear instructions on some minor details.

We also believe that the group worked adequately as we managed to successfully deploy the application. But sadly our coordination was not the best and we even ended up failing of the checkpoints due to a minor error. With this in mind we believe that one aspect in which we should improve is in the way we divide work in future projects.

Still, our solution is quite simple mostly because we do not do any changes to the codebase and focused on the basics. We only ensured a deployment and undeployment, we did not deliver the best possible one in terms of efficiency and response time, specially for multiple clients.